Semantics of UML Statecharts in PVS
Demissie B. Aredo
Norwegian Computing Center
P. O. Box 114 Blindern, N-0314 OSLO, Norway.
E-mail: aredo@nr.no

ABSTRACT
In this paper, we propose formal semantic definitions for UML statecharts in the specification language of PVS (Prototype Verification System). We also propose a general framework for translating UML statecharts into PVS specifications, and show how the resulting specification can be model-checked by using the PVS toolkits. This work is a part of a long-term vision to explore how the PVS tools could be used to underpin practical tools for analysis of UML models. It contributes to the ongoing effort on providing precise semantics definitions for UML notations with the aim of clarifying the language as well as supporting the development of semantically based tools.

Keywords: Formal Semantics, UML, PVS, Method Integration, Statecharts

1. INTRODUCTION
The Unified Modeling Language (UML) [24],[20] is an industrial standard for object-oriented modeling languages that was standardized by the Object Management Group (OMG). It is a collection of several description techniques which are suitable for modeling different aspects of software systems. Compared to other object-oriented modeling languages in software engineering, UML is more precisely defined and contains a great deal of formal specification notations, e.g. the use of Object Constraint Language (OCL) [28] for specifying constraint. However, semantic definitions for UML notations are not precise enough to support rigorous reasoning - a limitation that hampers its application to rigorous system development.

In this paper, we propose formal semantics the UML statecharts. Our aim is to achieve two goals. Firstly, we provide semantic model for basic modeling elements of UML statecharts using the PVS specification language [23],[21]. This consists of formal representation of the abstract syntax and the well-formedness rules, and model-checking the resulting specification. Secondly, we propose a general scheme for translating UML statecharts into PVS specifications. This results in semantic models that are amenable to rigorous analysis. Using PVS tools such as the theorem-prover and model-checker, we rigorous reason about the resulting semantics models.

Several works have been undertaken to provide mathematical basis to the concepts underlying object-oriented (OO) models using different approaches and semantic foundations. In general, formalization approaches can be categorized into three: [10]: supplemental, OO-extension and method-integration. In the supplemental approach informal modeling notations are replaced by more formal constructs (e.g. [19] is based on this approach and involves the LOTOS [13] and syntropy [2] notations) The OO-extension approach extends existing formal methods by OO features thus making them more compatible with the concepts of object-orientation, e.g. VDM++ [6], Z++ [15] and Object-Z [4]. Even though a rich body of formal notation results from supplemental and extension approaches, the resulting semantic domain is more complex and suffers from lack of tool support [7]. Moreover, users have to deal directly with a certain amount of formal artifacts. This is one of the major barriers for whole-scale utilization of formal methods due to their esoteric nature.

The method-integration [8],[10],[26] approaches makes OO notations more precise and amenable to rigorous analysis by integrating them with suitable formalism(s) [9]. It is a more workable and commonly used approach to formalization of OO modeling notations. The OO notation and a carefully chosen formalism, and their respective CASE tools are integrated allowing developers to manipulate the graphical artifacts they have created without having an in-depth knowledge about the formal specifications that are processed at the back-end [7]. Our work is also based on the method-integration approach to formalize UML statecharts using the specification language of PVS as underlying semantic foundation.

The rest of the paper is organized as follows: In Section 2., a brief overview of the PVS specification language is presented with emphasis put on concepts and notations that will be encountered in later sections. In Section 3., small concepts of UML statecharts are discussed. In Section 4., semantic definitions for the basic concepts of UML statecharts are proposed. Finally, in Section 5., we draw some conclusions and discuss future works.

2. THE PVS ENVIRONMENT
PVS [3],[21],[22] is a formalism for design and analysis of system specifications. It consists of a highly expressive specification language tightly integrated with a typechecker a theorem-prover, and other tools. A strength of PVS is its capacity to exploit the synergy between the specification language and its tools, e.g. the type-checker uses the theorem-prover. The theorem-prover allows construction of proofs interactively and run them automatically after minor changes.

The PVS specification language (PVS-SL) provides a very general semantic foundation. It is based on the classical strongly typed higher-order logic. Its type system contains basic types such as boolean, integer, real, and constructors for sets, tuples, records, and functions types. A record type consists of a finite set of fields of general structure R:TYPE = {# a_1 : T_1, . . . , a_n : T_n} where a_i's are accessor functions and T_i's are type expressions. Given a record r: R, a function application-like term a_i(r), is used to access the i'th field of r. Tuples have structures similar to records except that the order of fields is significant in the tuples. A function type is specified as F:TYPE = [D → R] and models a type of functions with domain
D and range R where D and R are type expressions. Given a type T, the type of sets of elements of T is specified using one of the constructs pred[T] or setof[T], each of which is a shorthand for the predicate \( T \rightarrow \text{bool} \). For a given set \( s: \text{setof}[T] \) and \( t:T \), membership of \( t \) in \( s \) is determined by evaluating the truth value of one of the following expressions: member(t,s) or s(t).

The PVS-SL also supports definition of Abstract Data Types (ADTs) - a useful mechanism that enriches type specification in PVS. The system of the PVS-SL has been augmented by predicate subtyping and dependent typing mechanisms and supports a richer type system than the classical higher-order logic [6]. Subtyping makes type-checking more powerful and allows stronger checks for consistency and invariance in a uniform manner [3]. Subtyping, however, renders type checking undecidable in which case the type-checker generates proof obligations called Type Correctness Conditions (TCC's). The user is required to discharge the TCCs interactively using the PVS theorem prover to verify that the specification is type-correct. A great deal of TCC's are discharged automatically, whereas more involved ones require interactive use of the theorem-prover. Predicate subtypes can be specified in two different ways. Given a type \( T \) and a predicate \( p \) on elements of \( T \), a predicate subtype of \( T \) with respect to \( p \) can be specified as either \( S: \text{TYPE} = \{ t:T \mid p(t) \} \) or \( S: \text{TYPE} = \{ p \} \). When the expression of the predicate is not explicitly given, we can specify \( S \) as uninterpreted subtype of \( T \), symbolically: \( S: \text{TYPE FROM} T \).

The PVS-SL provides primitives to perform inductive reasoning, rewriting, and model checking. These features simplify the proof process as mechanical aspects can be automated quite easily [14]. Specifications in PVS are organized into hierarchies of theories. A theory may contain type, variable, and constant declarations, definitions, axioms, and theorems. Modularity and reusability are captured by parameterized theories that specify generic elements that are instantiated by theory abbreviation construct. Predicates, usually known as assumptions, are used to constrain the parameters of a generic theory. PVS-SL includes a library of an extensive set of built-in constructs known as preludes, that provides several useful definitions and lemmas.

It is beyond the scope of this paper to present the PVS-SL in detail. For a complete and detailed discussion on PVS, interested reader may refer to [3],[21],[22].

3. BASICS OF UML STATECHARTS

UML statecharts [20] are primary modeling elements for construction of executable models that capture complex dynamic behavior of reactive systems. A statechart describes an abstract machine that defines a set of existence conditions, called states, a set of behaviors or actions that can be performed in each of those states, and a set of events that may cause state transitions according to a set of well-defined rules.

A statechart describes a model element in isolation in terms of its interaction with the rest of the world by responding to certain events. A response of an object to an event, and the action that may ensue as a result depend on the current state of the object and the event that occurs. This may possibly result in performance of an action and a transition into another state. An event may cause a fir-

![Fig. 1. UML statechart for an Account Class](image)

An example of a UML statechart diagram shown in Figure 1 specifies a complete life cycle of an account object. An account can be either in the debit or the credit state depending on the value of its attribute balance. The banking system allows customers to withdraw a given amount of fund in debit, subject to fixed fee \( f \), hence the introduction of the debit state of the account. When an object is in the debit state, deposit(a) is the only operation allowed. At junction \( p \), a guard condition \( [a>b\geq 0] \) is evaluated to check the amount against the balance \( b \). Note that the balance \( b \) is less than zero when the account is in the debit state, and hence the deposited amount must be compared to \( -b \). If the guard condition \( [a>b\geq 0] \) is true, the account is transformed into the credit state, otherwise it remains in the debit state. In any case, the balance is
updated by computing b:=b+a-t, where t is some constant fee charged when the account is in debit state. When an account object is in the credit state, the deposit(a) event increases its balance by, and leaves its state unaltered. An occurrence of a withdraw(a) event when the account is in credit state, may transform it into the debit state or leave it in the same state depending on the truth value of the guard condition (b-a=0) at junction q. In either case, the balance is updated with the result of b:=b-a.

4. FORMALIZATION OF UML STATECHARTS

In this section, we provide semantic definitions for UML statecharts by transforming them into appropriate entities in the PVS specification language. We encode the abstract syntax of UML statecharts, and associated well-formedness requirements. Note that the PVS-SL is used as underlying semantic foundation and not as a description language and hence users are not expected to have an in-depth knowledge about neither the PVS-SL nor its proof system. We define semantic models for statecharts using bottom-up approach, i.e. starting with semantic definitions of basic model elements such as states, transitions, events and actions we provide semantic definition for statecharts as an appropriate composition of semantic definitions of its components. We treat the informal semantics descriptions provided in UML version 1.3 standard document [20] as a requirement specification on which the formal semantic models will be based. Some constraints on UML models may involve dynamic information, e.g. the number of objects created could only be available during run time.

We specify a parameterised theory that defines a predicate on sets of elements of a type given as parameter of the theory. The predicate optional?() filters the empty set and singleton sets of elements of the type.

```plaintext
optional[T : TYPE ] : THEORY
BEGIN
z, y : VAR T; s : VAR set[T]
singleton?(s):bool< EXISTS(x:(s));
FORALL (y:(s)) : y\in
optional?(s):bool< (empty?(s) OR singleton?(s))
END optional
```

Given a type T and a set s of elements of T, (s) denotes a subtype of T containing exactly the elements of s. For every type (class in the UML vocabulary) involved in optional multiplicity, a new theory is instantiated from the generic theory optional with the type as a parameter using the PVS construct known as theory abbreviation. For instance, for the type T, a theory optional[T] is defined as an instance of theory optional. The expression optional[T].optional? provides access to the predicate optional?.

```plaintext
optional[T : TYPE ]
s : (optional[T].optional?)
```

4.1 Abstract Syntax of UML Statecharts

We start by representation of the notions of model element, action, signal, and operation as uninterpreted types in the PVS specification language. The `ModelElement` is a root class from which every class in UML metamodel inherits. The details of these model elements are intentionally avoided since such details are irrelevant at the level of abstraction we are working.

```plaintext
ModelElement : TYPE+
Action,Signal,Operation : TYPE FROM ModelElement
```

Next, we discuss notions of states, transitions and statecharts, and formally represent them.

**States:** A state is a specification of a snapshot of values of program variables or behavior of an object that satisfies some, usually implicit, invariant conditions. Objects of a given class that are in the same state have the same qualitative responses to an occurrence of the same event. That is, they react to events in the same way, and execute the same sequence of actions, and may undergo the same set of transitions (apart from non-determinism).

A state vertex is an abstraction of a node in a statechart diagram. In the UML meta-model, state is a direct subclass of the class `ModelElement` and hence we represent it as a subtype of the type `ModelElement`. In general, a state vertex can be a source and target of any number of transitions. In the record type `State`, the field `asModelElement` captures properties inherited from the super-class `ModelElement`.

```
StateVertex : TYPE FROM ModelElement
```

The class `StateVertex` can be specialized into the following four kinds of states: `State`, `PseudoState`, `StubState`, and `SyncState`. A synchronous state is used to synchronize concurrent regions of a state machine. Pseudo states are vertices in the state machine that are used to connect multiple transitions into a transition path. A stub state appears within a submachine to refer to the actual subvertex contained within the referenced state machine. A state may have an `entry` action - the first action that takes place when the state is entered, a set of `internal` transitions and associated actions, and an `exit` action - the last action that takes place when the state is exited.

Usually, an event that does not enable a transition is discarded. However, it is sometimes useful to keep this event waiting until the next state. A set of events to which a state machine does not react while it is in a given state is described as a set of "deferable" events - the field `deferable` captures a set of such events. Note that we declare variables only once and use them in the later sections.

```plaintext
T : TYPE ; x, y : VAR T; s : VAR set[T]
optionalAction : THEORY = optional[Action]
```

```plaintext
State:TYPE =
[# asStateVertex: StateVertex,
  entry: (optionalAction.optional?),
  doActivity: (optionalAction.optional?),
  exit: (optionalAction.optional?),
  deferable: setOf[Event]]
```

```plaintext
PseudoStateKind:TYPE= {initial,deepHist,join,
  shallowHist,fork,junction,choice}
PseudoState:TYPE= [# asStateVertex: StateVertex,
  pseudoKind: PseudoStateKind #]
StubState:TYPE= [# asStateVertex: StateVertex,
  refState: String #]
```
SystmState::TYPE = [# asSystmVertex: SystmVertex, bound: nat #]

The class State is further specialized into SimpleState, CompositeState, and FinalState which we represent as subtypes. A composite state can be concurrent or sequential.

v: VAR StateVertex
SimpleState: TYPE FROM State
FinalState: TYPE = {v | outgoing(v) = 0}
CompositeState: TYPE =
  [# asState : State, isConcurrent: bool, substate: fin_set[StateVertex] #]
container: [StateVertex -> CompositeState]

The container function returns the smallest composite state (if any) that contains a state vertex. The field isubstate captures the set of direct sub-states of a state. It is used to define the function subvertex() which returns the set of all sub-states of a given composite state. The subvertexInc() returns the set of substates of a state including the state itself. When applied to the top state of a state machine, subvertexInc() returns the set of all state vertices in the state machine by recursive application of isubstate() to the vertices.

contains(v,cs): bool = CompositeState(cs) \ member(v, isubstate(cs))
subvertex(cs): RECURSIVE setof[StateVertex] =
  \[\forall cs. isubstate(cs) \subseteq subvertex(cs),
   MEASURE (LAMBDA cs: isubstate(cs) \subseteq 0)
subvertexInc(cs): setof[StateVertex] =
  union([cs],subvertex(cs))

If an event is deferred in a given composite state, then it is deferred in any substate of that state. We add the axiom deferax given below to captures this notion.

v, v': VAR StateVertex; cs: VAR CompositeState
deferax: AXIOM (v cs subvertexInc(cs)) ⇒
  (isdeferrable(cs) ≤ isdeferrable(v))

Transitions: A transition in UML statecharts models a change in object behavior from one state to another state (not necessarily distinct) as a result of a response to a reception of an event. The set of transitions specifies a reaction of an object to events, or the action carried out by its methods in response to occurrence of the event. In other words, an object in a given state, called the source of transition, evolves into another state, called target state, when a specific event occurs and a guard condition is satisfied, and perform a sequence of actions.

A transition in a statechart may be labelled by a string of the form e|c|a, which means that the occurrence of event e, when the guard condition c is true, triggers the firing of the transition, as a result of which the object performs sequence of actions a. The UML standard [20] also allows triggerless transitions, called completion transitions. They have implicit triggers, i.e. completion event, which are generated when all transitions, entry actions and activities in the currently active state are completed.

To define semantics of a transition, we need the types Event, Action, and Guard, and instances of the theory optional instantiated with these types. Then, the notion of transition is captured by a record type with appropriate set of fields.

Event: TYPE FROM ModelElement
Guard: TYPE = [# asModelElement: ModelElement, expression: BooleanExpression #]
optionalEvent: THEORY = optional[Event]
optionalGuard: THEORY = optional[Guard]
optionalAction: THEORY = optional[Action]

Transition: TYPE =
  [# asModelElement: ModelElement, source: StateVertex, trigger: (optionalEvent.optional?), guard: (optionalGuard.optional?), effect: (optionalAction.optional?), target: StateVertex #]

We define some operations that specify associations between states and transitions. The functions incoming() and outgoing() defined on StateVertex return, respectively, the set of transitions entering and leaving the vertex. A transition connects exactly one source state and one target state, which are retrieved by applying the accessor functions source and target respectively, to the transition record.

incoming: [StateVertex -> setof[Transition]]
outgoing: [StateVertex -> setof[Transition]]

State Machines: A state machine can be described completely by a topology, i.e. a composite state at the root of the state containment hierarchy, and a set of transitions. Given the top state of a state machine and the set of its transitions, all the remaining states can be retrieved by traversing the state containment hierarchy starting at the top state. Application of the subvertexInc() function described above to the top state of a state machine returns the set of all state vertices in the state machine.

Semantics of a state machine is defined as a record type whose set of fields contain the top state vertex, and the set of transitions. Symbolically,

StateMachine: TYPE =
  [# asModelElement: ModelElement, top: StateVertex, transitions: setof[Transition], context: ModelElement #]

context: [StateMachine -> Context]

The function context() determines the model element whose behavior is captured by the state machine. A model element can be described by several state machines, but a given state machine describes at most one model element. The specification of function context() ensures that this requirement is fulfilled.

The SubmachineState defined below is a syntactical convenience that facilitates modularity and reuse, and is semantically equivalent to a composite state. It is a placeholder for a state machine that is referenced by another state machine. The submachine() function defined below determines the state machine for which a submachine state stands in a given composite state. The stateMachine()
function returns the state machine to which a transition belongs.

\textbf{SubmachineState} : TYPE FROM CompositeState
submachine: [SubmachineState, CompositeState→ StateMachine]
stateMachine : [Transition → StateMachine]

4.2 Well-formedness Requirements

In this section we formalize well-formedness requirements (WFRs) on some of the modeling elements described above. The well-formedness rules can be defined in the same theory as the model elements they constrain or in a separate theory and imported. We follow the latter option since this approach matches the informal descriptions given in the standard document of UML v1.3 [20]. The WFRs are labelled with the labels in the UML standard document [20] suffixed with the initial letter of the model element they constrain. For instance, rule\textit{CS1} corresponds to the first well-formedness rule for composite state.

\textbf{s} : VAR State; \textbf{c1} : VAR CompositeState
\textbf{v} : VAR StateVert; \textbf{m} : VAR StateMachine
\textbf{ps} : VAR PseudoState; \textbf{t} : VAR Transition

4.2.1 WFRs on Composite States:

The following WFRs apply to \textit{CompositeState}. A composite state can contain at most one vertex of each of the pseudostate of \textit{initial}, \textit{deepHist}, and \textit{shallowHist} kind.

\textit{ruleCS1}(cs) : bool =
optional?(ps\in subvertex(cs) ∧ pseudoKind(ps) = initial))
∧ optional?(ps\in subvertex(cs) ∧ ps deepenHist))
∧ optional?(ps\in subvertex(cs) ∧ pseudoKind(ps) = shallowHist))

A concurrent composite state must have at least two direct subvertices each of which is a composite state.

\textit{ruleCS2}(cs) : bool = isConcurrent(cs) ⇒
(\|subvertex(cs)\| ≥ 2) ∧ (subvertex(cs) ⊆ CompositeState)
where \|\| is a function that returns the cardinality of a set. A given state vertex can be a part of at most one composite state.

\textit{ruleCS3}(c) : bool =
(v\in substate(c) ∧ v\in substate(c1)) ⇒ c = c1

WFRs on Transitions: A fork segment should not have guards or triggers:

\textit{ruleT1}(t) : bool = (pseudoState(source(t)) ∧ pseudoKind(source(t)) = fork) ⇒
(guard(t) = 0 ∧ trigger(t) = 0)

A join segment should not have guards or triggers

\textit{ruleT2}(t) : bool = (pseudoState(target(t)) ∧ pseudoKind(target(t)) = join) ⇒
(guard(t) = 0 ∧ trigger(t) = 0)

A fork segment should always target a state:

\textit{ruleT3}(t) : bool = (stateMachine(t) ≠ 0 ∧ pseudoState(source(t)) ∧ pseudoKind(source(t)) = fork) ⇒
State(target(t))

A join segment should always originate from a state:

\textit{ruleT4}(t) : bool = (stateMachine(t) ≠ 0 ∧ pseudoState(target(t)) ∧ pseudoKind(target(t)) = join) ⇒
State(source(t))

Transitions outgoing from a pseudostates may not have a trigger:

\textit{ruleT5}(t) : bool = PseudoState(source(t)) ⇒
trigger(t) = 0

Join segments should originate from orthogonal states:

\textit{ruleT6}(t) : bool = (pseudoState(target(t)) ∧ pseudoKind(target(t)) = fork) ⇒
isConcurrent(source(t))

Fork segments should target orthogonal states:

\textit{ruleT7}(t) : bool = (pseudoState(source(t)) ∧ pseudoKind(source(t)) = fork) ⇒
isConcurrent(target(t))

An initial transition at the topmost level may have a trigger with the stereotype "create". An initial transition of a StateMachine modeling a behavioral feature has a CallEvent trigger associated with that BehavioralFeature. Apart from these cases, an initial transition never has a trigger:

\textit{CallEvent} : TYPE FROM Event
\textit{stereotype} : [ModelElement → ModelElement]
\textit{ruleT8}(t) : bool = (pseudoState(source(t)) ∧
kind(source(t) = initial)
⇒ (trigger(t) = 0)
∨ (container(source(t)) ∧
name(stereotype(trigger(t))) = "create")
∨ (BehavioralFeature(context(source(t))) ∧
CallEvent(trigger(t)) ∧
operation(trigger(t) = container(stateMachine(t))))

WFRs on State Machines: A state machine is aggregated either within a classifier or a behavioral feature. The context of a state machine should be an object or a behavior as specified by the well-formedness requirement \textit{ruleSM1} given below.

\textit{ruleSM1}(m) : bool = Classifier(context(m)) ∨
BehavioralFeature(context(m))

The top state of a state machine is always a composite state.

\textit{ruleSM2}(m) : bool = CompositeState(top(m))
The top state cannot have a container state.

\textit{ruleSM3}(m) : bool = container(top(m)) = 0
The top state cannot be the source of a transition.

\textit{ruleSM4}(m) : bool = outgoing(top(m)) = 0
If a state machine describes a behavioral feature, it contains no trigger of type CallEvent, apart from the trigger on the initial transition.

\textit{ruleSM5}(m) : bool = BehavioralFeature(context(m))
⇒ (\forall t: t\in transitions(m) ∧
NOT (pseudoState(source(t)) ∧
pseudoKind(source(t)) = fork) ⇒
trigger(t) = 0)
4.3 Semantic Definitions

Once the abstract syntax of basic elements of UML state machines, and well-formedness requirements are precisely encoded in the PVS specification language, providing semantic definitions for more complex model elements is easier. Formalizing semantics concepts of UML state machines paves a way for specifying important properties exhibited by the system and for rigorous reasoning about their correctness.

In general, for a UML model M, whose abstract syntax is encoded in the PVS-SE as SyntaxM and its well-formedness requirements as predicates rule1, ..., ruleMk, its semantics SemM is the predicate subtype of SyntaxM with respect to the conjunction of its well-formedness predicates. For instance, semantics of the state machine is defined as follows:

\[
\text{SemStateMachine: TYPE} = \{ \text{ruleSM1}(m) \land \text{ruleSM2}(m) \land \text{ruleSM3}(m) \land \text{ruleSM4}(m) \land \text{ruleSM5}(m) \}
\]

A state is said to be active when it is entered as a result of transition and becomes inactive when it is exited. A state can be thought of as a predicate on the set of program variables. The state is active when this predicate returns true. For a composite state that is active, and non-concurrent, exactly one of its substates is active. If a composite state is active and concurrent, then all of its substates are active.

active: [StateMachine -> bool]
activeAxi: AXiom (active(c) \land NOT isConcurrent(c) \land \forall v \in subvertex(c) \Rightarrow \|\langle v : (substate(c)) / active(v)\rangle\| = 1

activeAxi2: AXiom (active(c) \land isConcurrent(c) \land \forall v \in subvertex(c) \Rightarrow (FORALL \langle v : (substate(c)) \rangle : active(v))

If a given state is active, then every composite state containing the state, directly or transitively, is also active. Since some of the composite states may be concurrent, a current active state is represented by a tree of states, called state configuration, starting with the top most composite states down to individual simple states at the leaves.

configuration : [StateMachine \rightarrow \text{setof[StateMachine] configuration(sm) = \{ s | s \in subvertex(top(sm)) \land active(s)\}

More advanced semantic concepts such as conflicting transitions, firing priorities, etc. can similarly be formalized in terms of the basic concepts of UML statecharts defined above.

5. CONCLUSIONS

We have proposed a framework for semantic definition for UML statecharts using the PVS specification language as underlying semantics foundation. The main motivation for this work is to give a precise and equi-descriptive description of the UML statecharts. Such a precise description is required as a reference model for implementing tools for code generation, simulation and verification of UML statecharts. An implementation of our framework integrates a UML CASE tool and the PVS toolkit resulting in heterogeneous platform that combines the strengths of a semi-formal graphical modeling notation and a formal verification environment. Other benefits of transforming the UML statecharts into the PVS-SL include the ability to produce precise and analyzable specifications, and the availability of PVS toolkit that supports rigorous reasoning about the resulting semantic models.

Several semantics for statecharts have been proposed in the literature, e.g. [12],[11],[16],[18],[25],[27]. Most of them are concerned with defining semantics of the classical Harel’s statecharts [12]. For instance, Harel et al. [12],[11] presents semantics of classical statecharts in the STATEMATE system. Other researchers discuss formalization of UML statecharts based on hierarchical automata [18]. The representation in hierarchical automata is not well-suited for tool development [17]. It does not directly support transition across compound states, and the hierarchical structure must be flattened before using it in a model checker. The work presented in the sequel is similar to the work discussed in [27], yet this work is more detailed.

This work contributes to the ongoing effort to provide formal standard semantic definitions for UML notations, with the aim of clarifying and disambiguating the language as well as supporting the development of semantically based tools. It is a part of our long-term vision to explore how the PVS tool set could be used to underpin practical CASE tools to analyse UML models.

Acknowledgements

I would like to thank Prof. Ola Owe, Dr. Wenhui Zhang, and Dr. Jesa Traore for their comments that were invaluable for the improvement of the paper. This work was funded by the Research Council of Norway through the ADAPT-FT project.

References


